Low-Complexity Dynamic Translation in VDebug

Prashanth P. Bungale, Swaroop Sridhar, and Jonathan S. Shapiro
{ prash, swaroop, shap } @ cs.jhu.edu

Systems Research Laboratory
The Johns Hopkins University
Baltimore, MD 21218, U. S. A.

March 10, 2004

Abstract

Machine-level dynamic binary translation has been used in applications ranging from debugging, performance analysis, and security policy enforcement to full machine virtualization. Most implementations are optimized for performance rather that simplicity: they translate to an internal intermediate form before generating target code. While an intermediate form greatly assists certain types of code monitoring transformations, many applications do not require these transformations, and the performance arguments for an intermediate form appear questionable. In some applications, the need for translator simplicity outweighs the desire for translator generality.

VDebug is an x86 to x86 dynamic translation system designed to achieve least complexity rather than maximal performance. Originally designed as a supervisor-mode, bare-metal translator, Vdebug makes minimal assumptions about the supporting runtime environment, and makes no attempt to optimize guest code during translation. The resulting implementation exposes the overheads that are intrinsic to all binary translation mechanisms rather than specific to one or another set of optimizations. Surprisingly, VDebug yields performance comparable to more aggressive translation strategies. This paper focuses presents the key implementation techniques used in the user-mode and supervisor-mode translators.

1. Introduction

Binary translation techniques have been used for application level debugging and performance analysis, for security policy enforcement, and for full machine virtualization. Static binary rewriting tools such as *pixie* [1] have been used as the basis for comprehensive debugging and analysis tools such as SGI SpeedShop [2]. Valgrind [3] uses dynamic translation for performance measurement, memory analysis, and execution profiling. VMware uses supervisor-only dynamic translation for full machine emulation [4]. Work on "program shepherding" has been proposed as a techinque for enforcing security policies [5]. Modern dynamic translation techniques generally offers overheads in the general range of 5%. The DynamoRio system [6] is a reasonable current "gold standard" for dynamic translation.

While performance is an important goal for dynamic translators, it is not the *only* important goal. Run-time optimization comes with three significant costs:

- The dynamic translation system becomes intrinsically harder to maintain.
- The runtime cost of optimization offers diminishing returns as the cost of translation rises.

- The instruction cache footprint of the translator rises, leading to significant cost from cache residency competition. Simultaneously, the effective reuse rate of translator code falls as the specialization of the dynamic optimizer rises.

All of these points suggest that it is useful to have a reference dynamic translation system designed for least complexity. Such a design nominally accepts lower runtime performance for lower translation overhead, but the resulting performance may compare favorably with more complicated systems in practice – especially so when significant amounts of one-time startup code must be translated during application startup. A simpler translation system can serve as a reference implementation for use in validating more sophisticated implementations. VDebug is simple enough to be suitable for robust supervisor-mode implementation.

The VDebug translator is predicated on several observations:

- The majority of translated instructions are *innocuous*, meaning that they can be translated verbatim without any modification at all. This is true in the same way that most instructions are neither *sensitive* nor *privileged* in virtual machine emulators [7].
- Nearly all of the remaining instructions relate to management of control flow. Careful translation of these instructions is performance-critical, and the techniques for doing so have been underexplored in the existing literature.
- The few instructions that truly need non-trivial translations are (statistically speaking) never used. While gratuitous overhead should be avoided, the translated performance of these instructions isn't critical.
- The need for optimization is self-fulfilling. If aggressive optimization is eliminated, the entire guest register set can (with care) be preserved in the actual hardware registers. Any significant alteration of data references soon introduces a need to reserve a register for use by the translation system itself.

Based on these observations, we have implemented a simple dynamic translation system, and report on its results here.

2. Basic Mechanism

The basic translation mechanism of our system is similar to that of Dynamo [8], Mojo [9], or a number of other dynamic binary translators. *VDebug* proceeds by alternating its execution between "translation mode" and "target mode." During target mode, instructions are executed out of a region of memory known as the basic block cache. This region contains translated basic blocks (xBBs) resulting from the on-demand translation of guest basic blocks (gBBs). When the desired basic block cannot be found in the cache, execution switches to translation mode and the missing basic block is appended to the cache.

The switch from translation mode to target mode can be viewed as a specialized form of context switch: target state is unloaded from the processor into a temporary save area, the translator performs some translation, and execution resumes somewhere in the basic block cache. The defining issues in a translator of this form are:

1. Where does the definitive version of guest register state reside?

2. How is the term "basic block" defined? That is, what is the real unit of translation?
3. How are direct jumps managed?
4. How are indirect jumps (register, memory) handled?
5. How are call and return implemented?

2.1. Register State

The first issue in the design of a dynamic translator is to determine where the definitive copy of guest register state will be kept. Does guest register state generally reside in the hardware registers, or does it reside in some memory data structure to be loaded and spilled at need? On the Pentium, where registers are few, there is considerable incentive to preserve all available hardware registers for application use. If guest registers can be kept entirely in the hardware register set, the translator can eliminate the need for load and spill operations during basic block translation. This in turn eliminates the need to examine the registers used by normal instructions during translation. If guest registers cannot be preserved within the hardware register set, then a translator with intermediate code is effectively mandated.

Actually, there are two distinct cases that need to be considered for register management:

- Context switching from guest translated code to/from the translator.
- Spilling register state in order to look up indirect branch destinations.

The second is discussed below in Section 2.4. We address the first here.

In abstract, a dynamic translator requires a small amount of state to be shared between the guest and the translator – or more precisely, between the translated code and the translator. This state records the next guest PC (NPC) and the guest register save area. The save area is used to spill the guest register set when transferring control to the translator. The NPC value is used to tell the translator where translation (and therefore execution) should resume in the next execution phase. As a practical matter, these can be combined by generalizing the save area into a machine state (or Mstate) region that contains a reserved NPC slot in addition to the register save area.

VDebug uses two different implementations of MState depending on whether we are executing in user mode or supervisor mode. In user mode, we assume that the guest has a valid stack that is backed by a general-purpose page fault handler. When necessary, register values are spilled to this stack before control is transferred to the translator. The design rule is that *VDebug* cannot be observed to have changed the stack during execution of guest instructions. We assume implicitly that two threads of control in the same application address space do not examine each other's stacks.

When executing supervisor code, a memory-based MState area can unfortunately be detected by the guest code. For reasons discussed in Section 3, we are attempting to minimize the visibility of such state. We have chosen to implement register save for supervisor translation using the trap mechanism. To support this, we run guest supervisor code in non-privileged mode, simulating the "supervisor" bit of the page tables using shadow translation.

2.2. Unit of Translation

Compiler texts typically define a basic block (BB) as a sequence of code that is "bracketed" by branches. Every target of a control flow instruction (jump, branch, call, or return) is the beginning of a basic block, and every control flow instruction (jump, branch, call, or return) marks the end of a basic block. Dynamic translators and sophisticated optimizers also build on the notion of a *trace*, which is a sequence of instructions that (with high likelihood) will be executed in sequence. A basic block that is terminated by an unconditional jump to some other basic block is a single trace, because the two basic blocks are necessarily executed in sequence.

In a dynamic translator, there is a tension between two considerations:

- Space in the basic block cache is limited. While it is generally *correct* to translate a given basic block multiple times, it is desirable to avoid doing so where possible. This motivates a pure basic-block oriented approach in the interests of instruction cache reuse.
- Dynamic branch frequency is the defining characteristic in instruction performance. Where possible, it is desirable to assemble traces sequentially in the basic block cache, eliminating the intervening branches. This motivates translating traces wherever possible.

VDebug begins a new translation phase whenever it discovers that some instruction has not been translated. It proceeds with translation until it reaches any sort of control flow instruction. When a control flow instruction is encountered, an attempt is made to "continue the current trace" as described below in Section 2.3. That is, *VDebug* may translate several basic blocks during a single translation phase, doing its best to construct a trace sequentially in the basic block cache.

A key enabler of this optimization is that *VDebug* avoids adding preamble or postamble code to translated basic blocks. While such preambles and postambles are sometimes required, their use is rare. *VDebug* "outlines" them (places them outside the sequential control flow). In many cases this lets us entirely eliminate unconditional branches.

The *VDebug* instruction decoder is table-driven. After decode, the majority of instructions are "translated" by appending them unmodified into the basic block cache. If the basic block cache becomes full, *VDebug* discards the entire content of the basic block cache and resumes translation from nothing. While draconian, this policy allows branches to be patched within the basic block cache without maintaining the relocation infor-mation that would be required to unlink them selectively.

2.3. Direct Jumps

Direct jumps may be either simple or conditional. Where possible, *VDebug* eliminates simple jumps entirely. A check is made to determine whether the destination of the simple jump has already been translated (goal: basic block reuse). If so, then there is an existing xBB in the basic block cache; a jump to this xBB is emitted. Otherwise, the jump instruction is elided, a new xBB is started at corresponding to the gBB that is the destination of the original branch, and the target xBB is simply appended to the initial xBB in the basic block cache (the branch is elided). This forms a trace that is linearized on the assumption that mandatory branches are always taken.

Conditional jumps are somewhat trickier. The present *VDebug* implementation implicitly assumes that conditional branches are not taken, and that traces should be linearized on the

assumption that execution will proceed in a straight line. If the destination xBB of the conditional branch already exists in the trace cache, *VDebug* emits a conditional branch to that xBB directly. Otherwise, it begins a new xBB (because the next instruction might be a branch target), emits a conditional branch to a "fixup trampoline", and continues. This requires a temporary relocation record; fixup trampolines are not emitted until the current translation phase terminates. Once emitted, the conditional jump is patched to jump to the fixup trampoline.

The goal of the fixup trampoline is to allow the conditional branch to be patched in place once the destination xBB is translated. The fixup trampoline hand-saves some guest register state, performs a lookup for the target xBB, patches the original branch if the target xBB is found, or enters the translator if it is not. Note a general pattern here: the *VDebug* implementation prefers, when possible, to implement fixup-like mechanisms in code space rather than data space to avoid polluting the guest data cache. The mechanism used to discover the target address is shared in common with the indirect jump mechanism, described below.

It is unlikely that the current implementation is optimal; it is known, for example, that backward conditional jumps are likely-taken. We plan to examine the impact of changing this policy in future work; the goal in this generation is simplicity.

2.4. Indirect Jumps

Indirect jumps are those that proceed through either a register or a memory location. These are necessarily harder to translate than direct jumps, because the destination address cannot be known at dynamic translation time. The most common solution is to perform some form of hash-based lookup in a side data structure maintained by the translator. This solution requires a series of loads, compares, and conditional jumps that demand at least two registers (a data structure pointer and the NPC value) and the condition codes (which will be clobbered by the comparisons). Some scratchpad region is required. In user mode, *VDebug* exploits the stack to save the necessary temporaries.

Like Dynamo [8], *VDebug* uses a hash-based lookup scheme to translate indirect addresses. In contrast to Dynamo, the hash table entries are implemented in *code* rather than *data*. To perform an address resolution using this mechanism, *VDebug* first saves the condition code register (EFLAGS) and two scratch registers (%eax,%ebx) to the stack, computes a *hash* of the destination address, and performs a computed branch to the first entry in the hash chain:

```
pushf
push %eax
push %ebx
movl %eax, %rdest
movl %ebx, %rdest
andl $mask, %eax
addl $hash_base,%eax
jmp (%eax)
```

The hash chain itself performs a series of constant comparisons against known destination addresses and implements a conditional branch to the final destination basic block:

```
cmpl %ebx,$candidate-gNPC
je $xNPC-entry-trampoline
cmpl %ebx,%next-candidate-gNPC
```

```
je $next-xNPC-entry-trampoline
# fall-through:
jmp $need-translation-trampoline
```

The candidate trampoline restores the saved EAX, EBX, and flags registers and jumps to the target basic block. The main advantage of this code-based scheme is that it can be constructed in a way that guarantees effective I-cache utilization. It is also executed entirely using instruction stream references, and therefore does not consult the DTLB. While this is not a compelling advantage for user-mode code, it is important in the supervisor mode *VDebug* implementation.

We are considering a variant implementation in which each register jump instruction has a dedicated branch chain that is implemented as a binary tree. This would eliminate the need for both register saves and restores.

2.5. Call and Return

As in Dynamo, the `call` instruction is translated as a `push` (of the constant guest PC following the call instruction) followed by a `jmp` instruction. If the call is a register-based call instruction, the register jump trampoline is used as described above. This approach is sufficient, but it is not fast. The full description will make greater sense after we explain the translation of the return instruction.

The return instruction is a form of indirect branch that uses a value on the stack as its destination address. While the stack pointer must be modified as a side effect, this modification does not introduce significant new complexity. The problem is that call and return are much more frequent than register jumps, and we would dearly like to avoid the complex series of instructions that are needed to support the full register jump sequence. The full sequence is required to avoid exposing mutable state to a hostile guest, but we can do much better than this in the user-mode translator. For clarity, we will describe two schemes below: the naive scheme because it provides a sense of the technique, and the improved scheme that we will shortly be implementing.

The key constraint is that we must leave the original return address on the stack. There are a few rare programs (notably garbage collectors) that will behave erratically if the return address stored on the stack is not a valid user code address.

2.5.1. Naive Return Scheme

While we will need a hashed lookup scheme, we would like to ensure that the most recent return address appears first. *VDebug* accomplishes this by using a D-space hash table that is modified by every translated call instruction. The call instruction computes (at translation time) the guest address of the guest instruction after the call, computes the hash of this address, and stores the address of *the xBB instruction after the call* into this hash slot. The return instruction will recompute this hash at run time and branch *blindly and optimistically* to this address.

After translating the call instruction *VDebug* emits sanity check code that validates whether this destination is in fact the correct destination of the return instruction. This is done by comparing the guest return address value to the constant guest instruction address of the instruction

following the call. Should this compare fail, the jump register trampoline is invoked to transfer control the correct destination procedure.

The net effect of this hash table scheme is to serve as a form of stack cache in which the most recent call address for a given procedure is cached. This tends, in practice, to capture recursive calls well.

Note that perverse code is okay under this scheme. While the return instruction may not initially return to the correct destination, it is guaranteed to return to some destination consisting of code that *checks* whether the destination is correct. Any call-postamble will cause the right thing to happen eventually.

However, the naive scheme involves some unnecessary costs that we can eliminate: the scratch register save, the condition code save, and the hash computation at the return site.

2.5.2. Return Scheme NG

Our next generation return scheme relies on the fact that most calls are direct. Instead of smashing a hash location that is based on the return address, we smash a hash location that is based on the destination procedure start address. We then propagate the current procedure start address through the translator logic. The net effect is that we know the entry point of the current procedure at the time we translate the return instruction. This allows us to compute the hash lookup of the blind return address at compile time, replacing the return instruction with a jump instruction requiring no temporaries and no condition code save. The "pop" is performed at the return site after determining whether the condition codes remain live.

We then do more work in a per-call postamble. The most common instruction following a call is an `add` instruction (to pop the stack arguments). This instruction clobbers the condition codes, which tells us whether or not they need to be saved. The postable does a constant comparison against the value still resident on the stack and branches to recovery code if the return has transpired to the wrong address.

3. Virtualization of Supervisor-mode Code

When virtualizing the privileged state and instruction set of the x86 architecture, dynamic translation provides an effective way to simulate the behavior of sensitive and privileged instructions. An excellent discussion of privileged and sensitive instructions on the Pentium can be found in [7]. In *VDebug*, translation is performed by the emulator running in supervisor mode. Both guest supervisor and guest user code run in user mode on the underlying hardware; the supervisor protection bit in the page table is implemented by shadow page tables. Execution is initiated by a supervisor to user privilege transition, following which some number of basic blocks are executed before voluntarily trapping or otherwise faulting back into the emulator for some purpose. Though we have designed a full-fledged virtualization support for supervisor-mode code (in contrast to paravirtualization support, as provided by Denali[10] and Xen[11]), in this paper, we focus on three issues that proved particularly challenging.

3.1. Precise Interrupt / Exception Delivery

Dynamic translation has some major implications on interrupt/exception delivery to the guest. Specifically, a complication arises because of *pseudo-instruction-boundaries* being present in the translated instruction sequences, that were not present in the original (guest) instruction sequences. When a gBB instruction is translated into a sequence of multiple xBB instructions, new instruction boundaries have been introduced as far as the underlying hardware is concerned. When delivering an interrupt or an exception to the guest, it is imperative that we preserve the *atomicity* corresponding to guest instruction boundaries, and subsequently report the correct instruction boundary at which the interrupt or exception was delivered. Towards this, we have designed a framework that lays out the format of a translated instruction sequence corresponding to a given guest instruction, as follows:

1. Each individual instruction translation consists of a preamble, an “active” instruction, and a postamble.
2. Preamble and postamble are typically empty. If needed, a typical preamble contains register spill instructions to provide available temporary registers for use in instruction execution, or a push instruction (in the case of translating call), or a pop instruction (in the case of translating return), etc. The postamble restores the expected register values into the hardware registers for use by the following instruction. The important point to note about the requirements of a preamble and postamble is that a preamble can result in side-effects as long as its effects can be rolled back (i.e., can be safely undone); and, a postamble must not result in any side-effects as visible to the guest, but may result in effects visible to the emulator.
3. Most importantly, side-effects that are visible to the guest and are non-undoable, if any, can occur as a result of the execution of *only one* instruction (as seen by the underlying hardware), which we call the "*active*" instruction. Then, an interrupt or exception can occur only before or after that instruction, but not in between that instruction.

We now describe the exception delivery mechanism that uses the above framework: When an interrupt or exception occurs while the preamble part of a translation sequence (including the point just before the "active" instruction) is being executed, we roll back the execution up to the most recent instruction boundary. If it occurs after the execution of the "active" instruction, i.e., while the postamble is being executed, we report the next guest instruction boundary to the guest. Note that postamble operations invariably restore register values that will be restored anyway on return from supervisor mode, so postamble instructions do not need to be honored when a trap occurs.

In order to facilitate the rollback of the preamble's execution, the instruction generator generates *two* instruction streams in parallel. The first is a sequence of instructions that simulate the input basic block instructions. The second is a sequence of bytecodes that describe how to undo the effects of the preambles.

When an exception occurs in the preamble phase, these bytecodes are interpreted to determine which registers have been spilled and what modifications have been performed to the stack pointer during the preamble. These changes are undone, and execution will resume at the beginning of the preamble.

When an exception occurs during the postamble, the preamble bytecodes are consulted to determine which registers have already been spilled and what modifications to the stack may need to be undone. The trap handler is going to spill the registers anyway; it merely skips the spills of the registers that have already been spilled. The bytecodes encode sufficient information to know what stack operations must be undone.

Collectively, the bytecode system provides a form of instruction-level "reverse execution" that is sufficient to enable the appearance of instruction atomicity to be preserved.

3.2. Avoiding the occupation of guest's virtual address space for the emulator's use

One of the original *VDebug* goals was to be suitable for use in a virtual machine implementation. A desirable goal in a virtual machine is to implement a mechanism that cannot be detected overtly by the guest operating system. Modern processors generally implement split instructoin and data translation lookaside buffers (TLBs). With care, it is possible to ensure that the basic block cache never appears as a guest-accessable address by exploiting the fact that these two TLBs need not be consistent. A supervisor-mode implementation cannot rely on the guest stack as a valid spill area because it may not be valid. Unfortunately, a memory-based MState data structure is detectable by the guest as a read-write "hole" in their address space.

The virtualization of segment registers and the condition codes register (EFLAGS) requires the translated instruction sequences (i.e., xBB's) to have access to the virtual registers. If a trap into the emulator is to be avoided for this purpose, the xBB's should somehow have access to these virtual registers directly, but at the same time, the virtual register state must be protected from explicit tampering by the guest. For this purpose, the existing systems reserve a part of the guest's virtual address space for holding the virtual registers, among other similar state that needs to be accessible to the guest directly.

Supervisor-mode *VDebug* therefore exploits a fortunate circumstance of operating system code: operating systems do not generally use the SSE functional unit. The supervisor-mode "guest" does not normally include the SSE instructions or register set. When a register-indirect branch must be resolved, the necessary temporary registers are "spilled" by transferring them temporarily to the SSE functional unit registers. If needed by the guest OS, the SSE register set can be enabled and disabled through inserted traps to the emulator. This method allows the majority of guest execution to occur within the basic block cache even when a small number of "spills" are required.

3.3. Virtualization of Segmentation support

A final key issue arises in the management of segmentation. The problem is that an application can modify the global descriptor table and encounter a real hardware interrupt before the segment would normally be reloaded. *VDebug* uses a small, dedicated global descriptor table containing the six currently live descriptor values (in addition to the segment descriptors for the use of the emulator itself). Every time the guest performs a load into a segment register, the emulator loads a "tamed" copy of the descriptor table entry into the appropriate slot of the shadow descriptor table. This "*descriptor caching*" strategy resembles exactly what the hardware

does when it loads a segment register (i.e., the hardware actually updates the descriptor cache corresponsing to the register when it loads a segment register). This mechanism sharply contrasts with the exisiting "*descriptor table shadowing*" approaches, as done by Vmware [4], which require write-protecting the guest's GDT/LDT in order to catch updates to the descriptors that have currently been shadowed. Our mechanism is simpler to implement and also incurs lesser overhead (as we need not bother about catching updates to the tables by the guest).

References

[1] M.D. Smith. *Tracing with pixie*. Technical Report No. CSL-TR-91497, Computer Systems Laboratory, Stanford University, Stanford, CA. L.K. John et al. / Microprocessors and Microsystems 23 (1999) 537--551 550.

[2] Silicon Graphics. *SpeedShop User's Guide*. Silicon Graphics Inc., 1998.

[3] Julian Seward. *The Design and Implementation of Valgrind*. March 2003.

[4] Scott W. Devine, Edouard Bugnion, Mendel Rosenblum. *Virtualization system including a virtual machine monitor for a computer with a segmented architecture*. United States Patent # 6,397,242, May 28, 2002.

[5] V. Kiriansky, D. Bruening, and S. Amarasinghe. *Secure execution via program shepherding*. In 11th USENIX Security Symposium, Aug. 2002.

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. *An infrastructure for adaptive dynamic optimization*. In 1st International Symposium on Code Generation and Optimization (CGO-03), March 2003.

[7] John Scott Robin, Cynthia E. Irvine. *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. Proceedings of the 2000 USENIX Security Symposium, August 2000.

[8] V. Bala, E. Duesterwald, and S. Banerjia. *Dynamo: A transparent runtime optimization system*. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), June 2000.

[9] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. *Mojo: A dynamic optimization system*. In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), Dec. 2000.

[10] Andrew Whitaker, Marianne Shaw, Steven D. Gribble. *Scale and Performance in the Denali Isolation Kernel*. Proceedings of the 2002 Symposium on Operating Systems Design and Implementation, December 2002.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield. *Xen and the Art of Virtualization*. Proceedings of the 2003 Symposium on Operating Systems Principles, October 2003.